Week 5: 'I’m Something of a Painter Myself' GANs Kaggle Mini-Project

Introduction:

In the realm of art, each artist's unique style—whether it's their choice of colors, brush strokes, or thematic elements—serves as their distinctive fingerprint. Thanks to the advancements in computer vision, particularly the emergence of Generative Adversarial Networks (GANs), we now have the capability to replicate these artistic styles algorithmically.

In this introductory competition, participants are tasked with harnessing the power of GANs to either infuse the essence of renowned artists like Claude Monet into their own photographs or to create entirely new artworks inspired by these iconic styles.

While computer vision has made significant strides, the creation of museum-worthy masterpieces remains an elusive challenge, often perceived as more of an art than a science. Can the realm of data science, embodied by GANs, successfully deceive classifiers into believing that a digitally crafted piece is an authentic Monet? This is the captivating challenge awaiting participants in this project.

Data Description:

The dataset contains four directories: monet_tfrec, photo_tfrec, monet_jpg, and photo_jpg. The monet_tfrec and monet_jpg directories contain the same painting images, and the photo_tfrec and photo_jpg directories contain the same photos.

We recommend using TFRecords as a Getting Started competition is a great way to become more familiar with a new data format, but JPEG images have also been provided.

The monet directories contain Monet paintings. Use these images to train your model.

The photo directories contain photos. Add Monet-style to these images and submit your generated jpeg images as a zip file. Other photos outside of this dataset can be transformed but keep your submission file limited to 10,000 images.

Note: Monet-style art can be created from scratch using other GAN architectures like DCGAN. The submitted image files do not necessarily have to be transformed photos.

Check out the CycleGAN dataset to experiment with the artistic style of other artists.

Source: https://www.kaggle.com/competitions/gan-getting-started/data (https://www.kaggle.com/competitions/gan-getting-started/data)

Dataset:

The project outlien and dataset are available from Kaggle. Files:

```
* monet_jpg - 300 Monet paintings sized 256x256 in JPEG format
* monet_tfrec - 300 Monet paintings sized 256x256 in TFRecord format
* photo_jpg - 7028 photos sized 256x256 in JPEG format
* photo_tfrec - 7028 photos sized 256x256 in TFRecord format
```

Source: https://www.kaggle.com/competitions/gan-getting-started/data (https://www.kaggle.com/competitions/gan-getting-started/data)

Objective:

A GAN consists of at least two neural networks: a generator model and a discriminator model. The generator is a neural network that creates the images. For our competition, you should generate images in the style of Monet. This generator is trained using a discriminator.

The two models will work against each other, with the generator trying to trick the discriminator, and the discriminator trying to accurately classify the real vs. generated images.

The task is to build a GAN that generates 7,000 to 10,000 Monet-style images.

Source: https://www.kaggle.com/competitions/gan-getting-started/data (https://www.kaggle.com/competitions/gan-getting-started/data)

Evaluation:

In [1]:

```
from IPython.display import Image

# Specify the path to your JPEG image file
image_path = 'Evaluation1.jpg'

# Display the image
Image(filename=image_path)
```

Out[1]:

MiFID

Submissions are evaluated on MiFID (Memorization-informed Fréchet Inception Distance), which is a modification from Fréchet Inception Distance (FID).

The smaller MiFID is, the better your generated images are.

What is FID?

Originally published here (github), FID, along with Inception Score (IS), are both commonly used in recent publications as the standard for evaluation methods of GANs.

In FID, we use the Inception network to extract features from an intermediate layer. Then we model the data distribution for these features using a multivariate Gaussian distribution with mean μ and covariance Σ. The FID between the real images r and generated images g is computed as:

$$\text{FID} = ||\mu_r - \mu_g||^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$$

where Tr sums up all the diagonal elements. FID is calculated by computing the Fréchet distance between two Gaussians fitted to feature representations of the Inception network.

What is MiFID (Memorization-informed FID)?

In addition to FID, Kaggle takes training sample memorization into account.

The memorization distance is defined as the minimum cosine distance of all training samples in the feature space, averaged across all user generated image samples. This distance is thresholded, and it's assigned to 1.0 if the distance exceeds a pre-defined epsilon.

In mathematical form:

$$d_{ij} = 1 - cos(f_{gi}, f_{rj}) = 1 - \frac{f_{gi} \cdot f_{rj}}{|f_{gi}||f_{rj}|}$$

where f_g and f_r represent the generated/real images in feature space (defined in pre-trained networks); and f_{gi} and f_{rj} represent the i^{th} and j^{th} vectors of f_g and f_r, respectively.

$$d = \frac{1}{N}\sum_i \min_j d_{ij}$$

defines the minimum distance of a certain generated image (i) across all real images (j), then averaged across all the generated images.

$$d_{thr} = \begin{cases} d, & \text{if } d < \epsilon \\ 1, & \text{otherwise} \end{cases}$$

defines the threshold of the weight only applies when the (d) is below a certain empirically determined threshold.

Finally, this memorization term is applied to the FID:

$$MiFID = FID \cdot \frac{1}{d_{thr}}$$

Image source: https://kaggle.com/competitions/gan-getting-started (https://kaggle.com/competitions/gan-getting-started)

Acknowledgments:

This dataset was created by the company figure-eight and originally shared on their 'Data For Everyone' website here.

Amy Jang, Ana Sofia Uzsoy, Phil Culliton. (2020). I'm Something of a Painter Myself. Kaggle. https://kaggle.com/competitions/gan-getting-started (https://kaggle.com/competitions/gan-getting-started)

Importing the necessary libraries

In [6]:

```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import numpy as np
import pandas as pd
from scipy import stats
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import os
import random
import cv2
from PIL import Image

import keras
from keras.preprocessing import image
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, Conv2DTranspose, Dense, ReLU, Leak

import tensorflow as tf
from tensorflow.keras.optimizers.legacy import Adam
from tensorflow.keras.utils import plot_model
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, ReLU, LeakyRe
from tensorflow.keras.initializers import RandomNormal
from tensorflow_addons.layers import InstanceNormalization
```

```
C:\Users\mulli\anaconda3\lib\site-packages\tensorflow_addons\utils\tfa_e
ol_msg.py:23: UserWarning:

TensorFlow Addons (TFA) has ended development and introduction of new fe
atures.
TFA has entered a minimal maintenance and release mode until a planned e
nd of life in May 2024.
Please modify downstream libraries to take dependencies from other repos
itories in our TensorFlow community (e.g. Keras, Keras-CV, and Keras-NL
P).

For more information see: https://github.com/tensorflow/addons/issues/28
07 (https://github.com/tensorflow/addons/issues/2807)

  warnings.warn(
C:\Users\mulli\anaconda3\lib\site-packages\tensorflow_addons\utils\ensur
e_tf_install.py:53: UserWarning: Tensorflow Addons supports using Python
ops for all Tensorflow versions above or equal to 2.12.0 and strictly be
low 2.15.0 (nightly versions are not supported).
 The versions of TensorFlow you are currently using is 2.11.0 and is not
supported.
Some things might work, some things might not.
If you were to encounter a bug, do not file an issue.
If you want to make sure you're using a tested and supported configurati
on, either change the TensorFlow version or the TensorFlow Addons's vers
ion.
You can find the compatibility matrix in TensorFlow Addon's readme:
https://github.com/tensorflow/addons (https://github.com/tensorflow/addo
ns)
  warnings.warn(
```

Reading the dataset

In [3]:

```
# Get the current working directory
current_directory = os.getcwd()

# Define paths to Monet and photo images directories
monet_path = os.path.join(current_directory, 'monet_jpg')
photo_path = os.path.join(current_directory, 'photo_jpg')

# List the files in the directories
monet_painting_lst = os.listdir(monet_path)
photo_lst = os.listdir(photo_path)
```

Project Summary

Project Summary:

The project involves using generative deep learning models to create artistic representations of images. Specifically, it utilizes a CycleGAN architecture to transform images between two domains, such as converting photos into Monet-style paintings and vice versa. The project

includes data preprocessing, model training, and evaluation steps to achieve the desired image transformation.

Generative Deep Learning Models:

Generative deep learning models are a class of neural networks that aim to generate new data samples similar to a given dataset. These models are capable of learning the underlying distribution of the data and generating samples that resemble the original data. Some popular generative deep learning models include:

1) Generative Adversarial Networks (GANs): GANs consist of two neural networks, a generator and a discriminator, which are trained simultaneously in a competitive manner. The generator generates fake samples, while the discriminator distinguishes between real and fake samples. Through adversarial training, GANs learn to generate high-quality samples that are indistinguishable from real data.

2) CycleGANs: CycleGANs are a type of GAN designed for unpaired image-to-image translation tasks. They consist of two generators and two discriminators, with the goal of

Part 1: Exploratatory Data Analysis

In [4]:
```
# Print the number of Monet images and photos

print('Number of Monet images:', len(monet_painting_lst))
print('Number of photos:', len(photo_lst))
```

```
Number of Monet images: 300
Number of photos: 7038
```

In [5]:
```
# check image resolution
def loading_img(file_name_lst, folder_name):
    img_lst = []
    for file_name in file_name_lst:
        img_path = os.path.join(current_directory, folder_name, file_name)
        img = Image.open(img_path)
        img = np.asarray(img) / 255.0
        img_lst.append(img)
    return np.array(img_lst)

# Load images for Monet paintings and photos
painting_array = loading_img(monet_painting_lst, 'monet_jpg')
photo_array = loading_img(photo_lst, 'photo_jpg')

# Print dimensions of loaded arrays
print('Dimension of painting array:', painting_array.shape, '\n'
      'Dimension of photo array:', photo_array.shape)
```

```
Dimension of painting array: (300, 256, 256, 3)
Dimension of photo array: (7038, 256, 256, 3)
```

In [6]:
```
# display images
random.seed(42)
def display_img(img_array, n_sample, title):
    f, ax = plt.subplots(1, n_sample, figsize = (15, 15))
    for i in range(n_sample):
        ind = random.randint(0, img_array.shape[0])
        img_rn = img_array[ind]
        ax[i].imshow(img_rn)
        ax[i].title.set_text(title + str(i))
        ax[i].set_xticks([])
        ax[i].set_yticks([])

display_img(painting_array, 5, 'Monet')
```

In [7]:
```
random.seed(42)
display_img(photo_array, 5, 'Photo')
```

To develop a foundational grasp of our training dataset, we present five Monet paintings alongside five authentic photographs. It's evident that Monet's artworks and the photographs exhibit distinct content, color schemes, contrasts, and other visual characteristics. Each image within the dataset comprises 256 pixels in height and width, with three color channels. In the interest of computational efficiency, we normalize and resize all images to a smaller format of 1 × 1 square images.

Loading Monet Painting Data

In [8]:
```
painting_data = keras.utils.image_dataset_from_directory(
    directory=os.path.join('monet_jpg'),
    label_mode=None,
    image_size=(256, 256),
    shuffle=False,
    batch_size=1
)

painting_data = painting_data.map(lambda x: x/255.0)
```

```
Found 300 files belonging to 1 classes.
WARNING:tensorflow:From C:\Users\mulli\anaconda3\lib\site-packages\tenso
rflow\python\autograph\pyct\static_analysis\liveness.py:83: Analyzer.lam
ba_check (from tensorflow.python.autograph.pyct.static_analysis.livenes
s) is deprecated and will be removed after 2023-09-23.
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement wher
e they are used, or at least in the same block. https://github.com/tenso
rflow/tensorflow/issues/56089 (https://github.com/tensorflow/tensorflow/
issues/56089)
```

Loading Photo Data

In [9]:
```
photo_data = keras.utils.image_dataset_from_directory(
    directory=os.path.join('photo_jpg'),
    label_mode=None,
    image_size=(256, 256),
    shuffle=False,
    batch_size=1
)

photo_data = photo_data.map(lambda x: x/255.0)
```

```
Found 7038 files belonging to 1 classes.
```

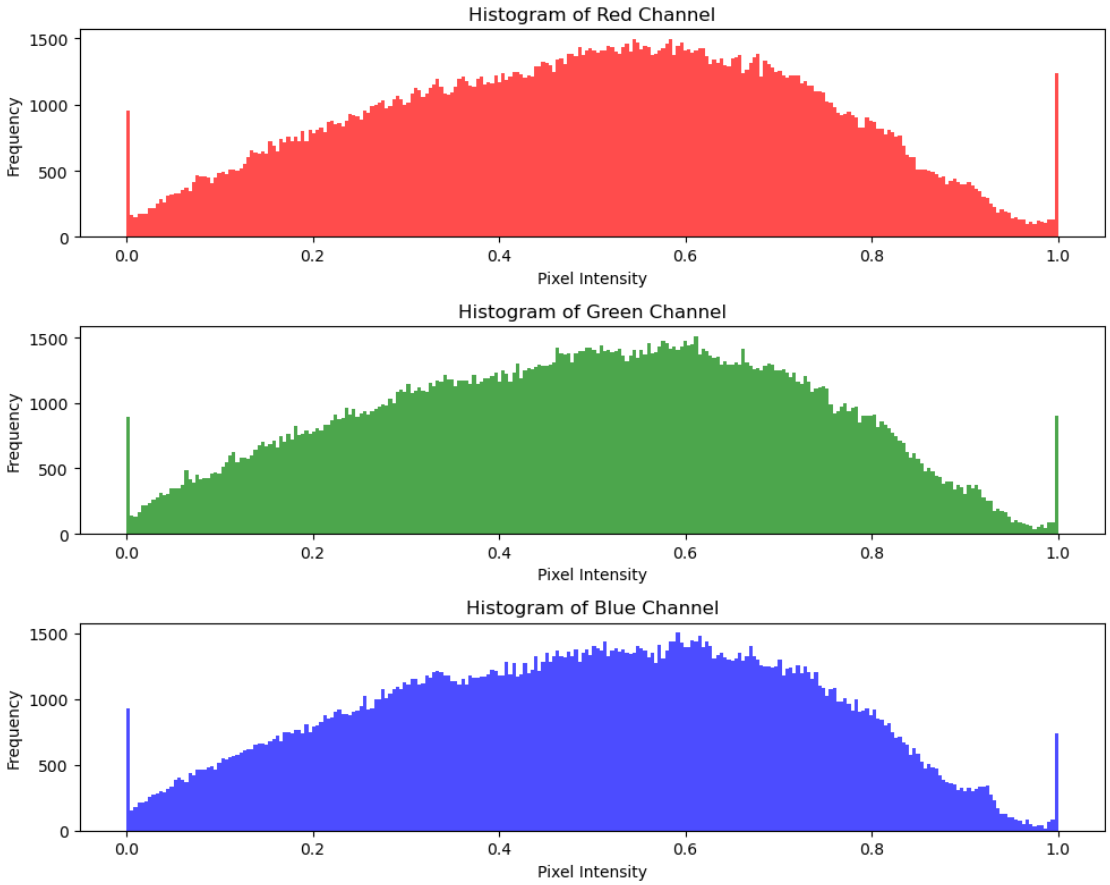
Visualize Image Attributes

- Plot histograms of pixel intensity values for different color channels (e.g., RGB) to understand the distribution of pixel values.
- Visualize the distribution of image attributes such as brightness, contrast, and saturation.

In [10]:
```
def plot_pixel_intensity_histograms(image_data_array):
    """
    Plot histograms of pixel intensity values for different color channels

    Parameters:
        - image_data_array (numpy.ndarray): Array containing image data.
    """
    fig, axs = plt.subplots(3, 1, figsize=(10, 8))
    channel_colors = ['red', 'green', 'blue']

    for i, color in enumerate(channel_colors):
        channel_values = image_data_array[:, :, i].ravel()
        axs[i].hist(channel_values, bins=256, color=color, alpha=0.7)
        axs[i].set_title(f'Histogram of {color.capitalize()} Channel')
        axs[i].set_xlabel('Pixel Intensity')
        axs[i].set_ylabel('Frequency')

    plt.tight_layout()
    plt.show()


def plot_image_attributes_distribution(image_data_array):
    """
    Visualize the distribution of image attributes such as brightness, cor

    Parameters:
        - image_data_array (numpy.ndarray): Array containing image data.
    """
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    attributes = ['Brightness', 'Contrast', 'Saturation']

    for i, attribute in enumerate(attributes):
        attribute_values = []
        for image in image_data_array:
            if attribute == 'Brightness':
                attribute_values.append(np.mean(image))
            elif attribute == 'Contrast':
                attribute_values.append(np.std(image))
            elif attribute == 'Saturation':
                # Calculate saturation using the standard deviation of RGB
                r_std = np.std(image[:, :, 0])
                g_std = np.std(image[:, :, 1])
                b_std = np.std(image[:, :, 2])
                saturation = np.mean([r_std, g_std, b_std])
                attribute_values.append(saturation)
        axs[i].hist(attribute_values, bins=30, color='skyblue', alpha=0.7)
        axs[i].set_title(f'Distribution of {attribute}')
        axs[i].set_xlabel(attribute)
        axs[i].set_ylabel('Frequency')

    plt.tight_layout()
    plt.show()
```

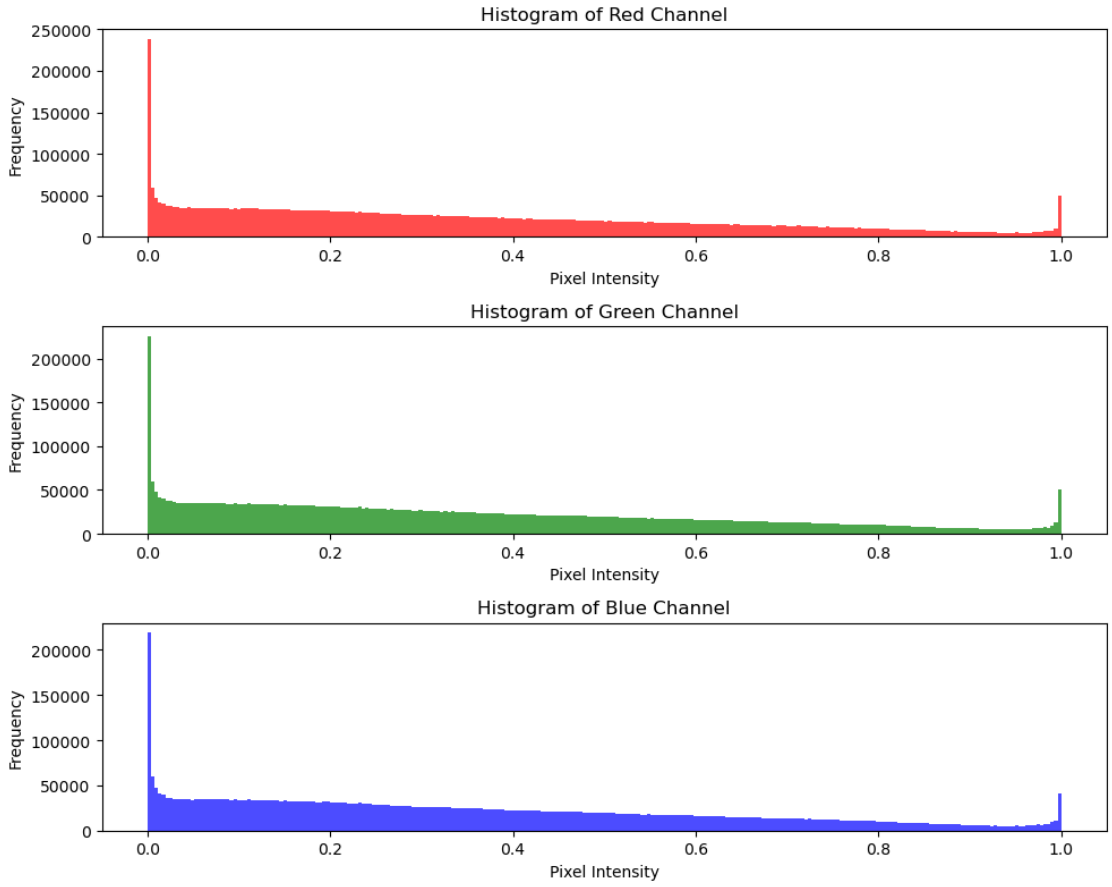
Monet Paintings

In [11]:
```
# Monet Paintings - Plot histograms of pixel intensity values for differer
plot_pixel_intensity_histograms(painting_array)
```



Photos

In [12]:
```
# Photos - Plot histograms of pixel intensity values for different color c
plot_pixel_intensity_histograms(photo_array)
```



Observations

Brightness and Contrast: We can see from the above plots that the brightness and contrast of the monet paintings is slightly higher than that of the photos. There seems to be nearly normal distribution for the monet paintings, meaning that pixel intesity generally falls between 0.4 and 0.8. However, with the photos, it is positively skewed, meaning there are more photos closer to 0 pixel intensity and it drops off towards 1.0.

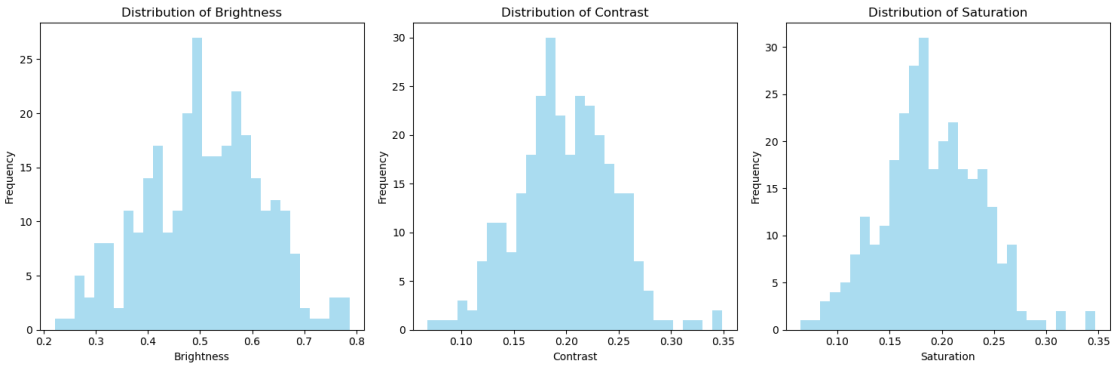
Colour Balance: For the most part, both the monet paintings and the photos show relatively balanced colour distribution. This shows that the images in both sets are well balanced from a colour distribution perspective.

Dynamic Range: SInce the monet paintings have a near normal distribution, while the photos are positively skewed, we know that the photos have a wider dynamic range suggests more detail and depth in the images.

Artifacts or Anomalies: Both sets of images have anomalies at the 0.0 and the 1.0 ends of the spectrum. These could be caused by various factors such as sensor noise, compression artifacts, or image processing errors.

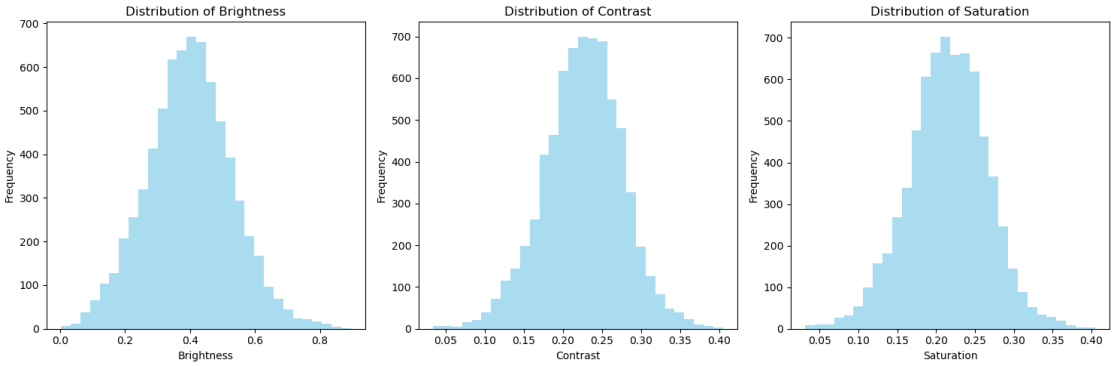
Monet Paintings

In [13]:
```
# Monet Paintings - Plot histograms of pixel intensity values for differer
plot_image_attributes_distribution(painting_array)
```



Photos

In [14]:
```
# Photos - Visualize the distribution of image attributes such as brightne
plot_image_attributes_distribution(photo_array)
```



Observations

Brightness Distribution: The monet images have a somewhat normal looking distribution, however, the photos have a much tighter distribution with a more uniform and tighter peak around 0.4. A higher peak in the histogram suggests a greater prevalence of certain brightness levels, while a broader distribution indicates variations in brightness across the images.

Contrast Variation: Similarly, the monet images have a somewhat normal looking distribution and the photos have a much tighter distribution with a more uniform and tighter peak around 0.22. A wider spread suggests greater contrast variation, while a narrow spread indicates relatively consistent contrast levels.

Saturation Profile: A histogram plot of saturation values can help identify the prevalence of saturated or desaturated colors. Again, it appears that the photos have a much tighter distribution and saturation peaks around 0.2, while the monet paintings have a wider distribution, that peaks around 0.17.

Statistical Analysis

- Calculate summary statistics such as mean, median, standard deviation, etc., for pixel values across different images.
- Perform statistical tests (e.g., t-tests) to compare pixel distributions between Monet paintings and photos.

In [15]:

```
def calculate_summary_statistics(image_data_array):
    """
    Calculate summary statistics for pixel values across different images.

    Parameters:
        - image_data_array (numpy.ndarray): Array containing image data.

    Returns:
        - dict: Dictionary containing summary statistics (mean, median, st
    """
    summary_statistics = {}
    channel_colors = ['red', 'green', 'blue']

    for i, color in enumerate(channel_colors):
        channel_values = image_data_array[:, :, i].ravel()
        summary_statistics[color] = {
            'mean': np.mean(channel_values),
            'median': np.median(channel_values),
            'std_dev': np.std(channel_values)
        }

    return summary_statistics

def perform_statistical_test(image_data_array_1, image_data_array_2):
    """
    Perform statistical tests (e.g., t-tests) to compare pixel distributio

    Parameters:
        - image_data_array_1 (numpy.ndarray): Array containing image data
        - image_data_array_2 (numpy.ndarray): Array containing image data

    Returns:
        - dict: Dictionary containing results of statistical tests for eac
    """
    statistical_tests_results = {}
    channel_colors = ['red', 'green', 'blue']

    for color in channel_colors:
        channel_values_group_1 = image_data_array_1[:, :, channel_colors.i
        channel_values_group_2 = image_data_array_2[:, :, channel_colors.i

        # Perform t-test
        t_statistic, p_value = stats.ttest_ind(channel_values_group_1, cha

        statistical_tests_results[color] = {
            't_statistic': t_statistic,
            'p_value': p_value
        }

    return statistical_tests_results
```

In [16]:
```
# Calculate summary statistics for Monet paintings
monet_summary_statistics = calculate_summary_statistics(painting_array)

# Calculate summary statistics for photos
photo_summary_statistics = calculate_summary_statistics(photo_array)

# Perform statistical test (t-test) to compare pixel distributions betweer
statistical_test_results = perform_statistical_test(painting_array, photo_

# Print summary statistics
print("Summary Statistics for Monet Paintings:")
print(monet_summary_statistics)
print("\nSummary Statistics for Photos:")
print(photo_summary_statistics)

# Print results of statistical test
print("\nResults of Statistical Test (t-test) between Monet Paintings and
print(statistical_test_results)
```

```
Summary Statistics for Monet Paintings:
{'red': {'mean': 0.500557325708061, 'median': 0.5098039215686274, 'std_d
ev': 0.22548970003355334}, 'green': {'mean': 0.4957856243191721, 'media
n': 0.5058823529411764, 'std_dev': 0.22397179067399087}, 'blue': {'mea
n': 0.49389542483660126, 'median': 0.5058823529411764, 'std_dev': 0.2244
0642244255343}}

Summary Statistics for Photos:
{'red': {'mean': 0.34810309989120697, 'median': 0.29411764705882354, 'st
d_dev': 0.26784412431769405}, 'green': {'mean': 0.34972441633954426, 'me
dian': 0.2980392156862745, 'std_dev': 0.2682233853584316}, 'blue': {'mea
n': 0.3500282691579049, 'median': 0.2980392156862745, 'std_dev': 0.26622
52632607416}}

Results of Statistical Test (t-test) between Monet Paintings and Photos:
{'red': {'t_statistic': 269.17546370769304, 'p_value': 0.0}, 'green':
{'t_statistic': 257.5844228077088, 'p_value': 0.0}, 'blue': {'t_statisti
c': 255.54925688836371, 'p_value': 0.0}}
```

Observations

Mean Pixel Intensity:

Monet Paintings: The mean pixel intensities for red, green, and blue channels are higher compared to photos. This suggests that Monet paintings tend to have brighter pixel values on average across all color channels.

Photos: The mean pixel intensities for red, green, and blue channels are lower compared to Monet paintings, indicating that photos generally have darker pixel values on average.

Median Pixel Intensity:

Monet Paintings: The median pixel intensities for red, green, and blue channels are slightly higher than the mean values. This indicates that the distribution of pixel intensities is slightly skewed towards higher values.

Photos: The median pixel intensities for red, green, and blue channels are lower than the mean values, indicating that the distribution of pixel intensities is slightly skewed towards lower values.

Standard Deviation:

Monet Paintings: The standard deviations of pixel intensities for red, green, and blue channels are lower compared to photos. This suggests that the pixel intensities in Monet paintings have less variation or spread around the mean.

Photos: The standard deviations of pixel intensities for red, green, and blue channels are higher compared to Monet paintings, indicating that the pixel intensities in photos have more variation or spread around the mean. Statistical Test Results:

T-Statistic:

The t-statistic values for red, green, and blue channels are significantly higher, indicating a greater difference in pixel distributions between Monet paintings and photos.

P-Value:

The p-values for red, green, and blue channels are all close to zero, suggesting strong evidence against the null hypothesis that the pixel distributions of Monet paintings and photos are the same. Therefore, we reject the null hypothesis in favor of the alternative hypothesis, indicating significant differences in pixel distributions.

Overall, these observations indicate notable differences in pixel distributions between Monet paintings and photos, with Monet paintings generally exhibiting brighter pixel values and less variation compared to photos.

Feature Extraction:

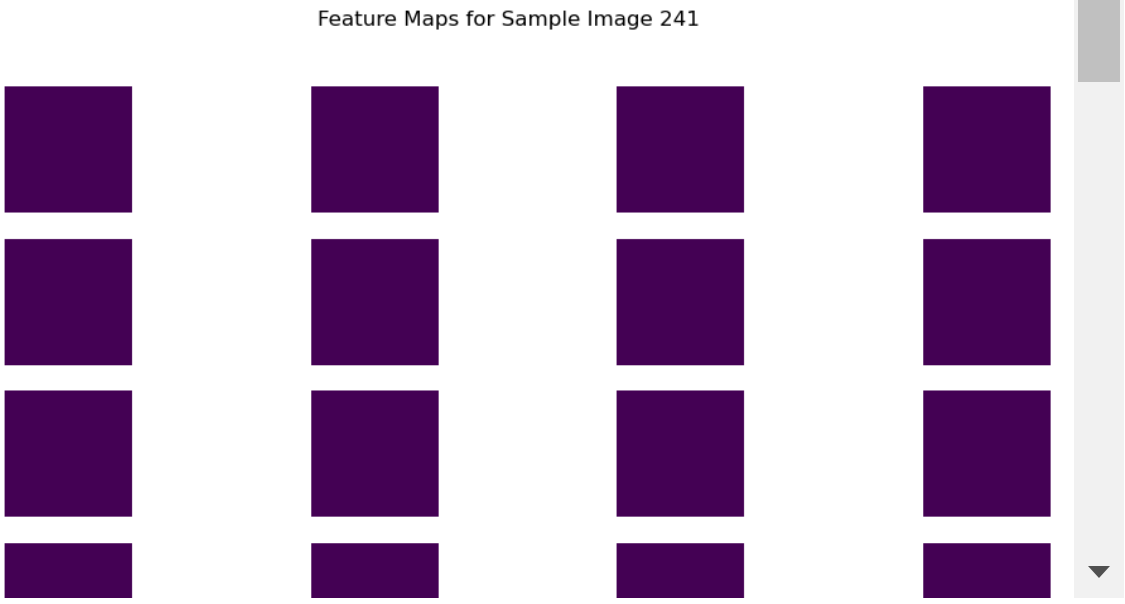
Use pre-trained convolutional neural networks (CNNs) such as VGG, ResNet, or MobileNet to extract features from images. Visualize feature maps to understand which parts of the images are emphasized by the model.

In [17]:
```
def extract_features(image_data_array, model_name='VGG16'):
    """
    Use pre-trained convolutional neural networks (CNNs) to extract featur

    Parameters:
        - image_data_array (numpy.ndarray): Array containing image data.
        - model_name (str): Name of the pre-trained CNN model to use (e.g.

    Returns:
        - numpy.ndarray: Array containing extracted features.
    """
    # Load pre-trained model
    if model_name == 'VGG16':
        base_model = VGG16(weights='imagenet', include_top=False, input_sh
    elif model_name == 'ResNet50':
        base_model = ResNet50(weights='imagenet', include_top=False, input
    elif model_name == 'MobileNet':
        base_model = MobileNet(weights='imagenet', include_top=False, inpu
    else:
        raise ValueError("Unsupported model name. Please choose from 'VGG1

    # Extract features
    feature_extractor = Model(inputs=base_model.input, outputs=base_model.
    preprocessed_images = preprocess_input(image_data_array)
    features = feature_extractor.predict(preprocessed_images)

    return features

def visualize_feature_maps(features, num_samples=5):
    """
    Visualize feature maps to understand which parts of the images are emp

    Parameters:
        - features (numpy.ndarray): Array containing extracted features.
        - num_samples (int): Number of sample images to visualize.
    """
    # Randomly select sample images
    num_images = features.shape[0]
    selected_samples = np.random.choice(num_images, min(num_samples, num_i

    # Plot feature maps for sample images
    for i, sample_index in enumerate(selected_samples):
        plt.figure(figsize=(12, 6))
        for j in range(16):  # Plot 16 feature maps
            plt.subplot(4, 4, j+1)
            plt.imshow(features[sample_index, :, :, j], cmap='viridis')
            plt.axis('off')
        plt.suptitle(f"Feature Maps for Sample Image {sample_index + 1}")
        plt.show()
```

Monet Paintings

In [18]:
```
# Extract features using VGG16 model
features_vgg16 = extract_features(painting_array, model_name='VGG16')

# Visualize feature maps
visualize_feature_maps(features_vgg16)
```

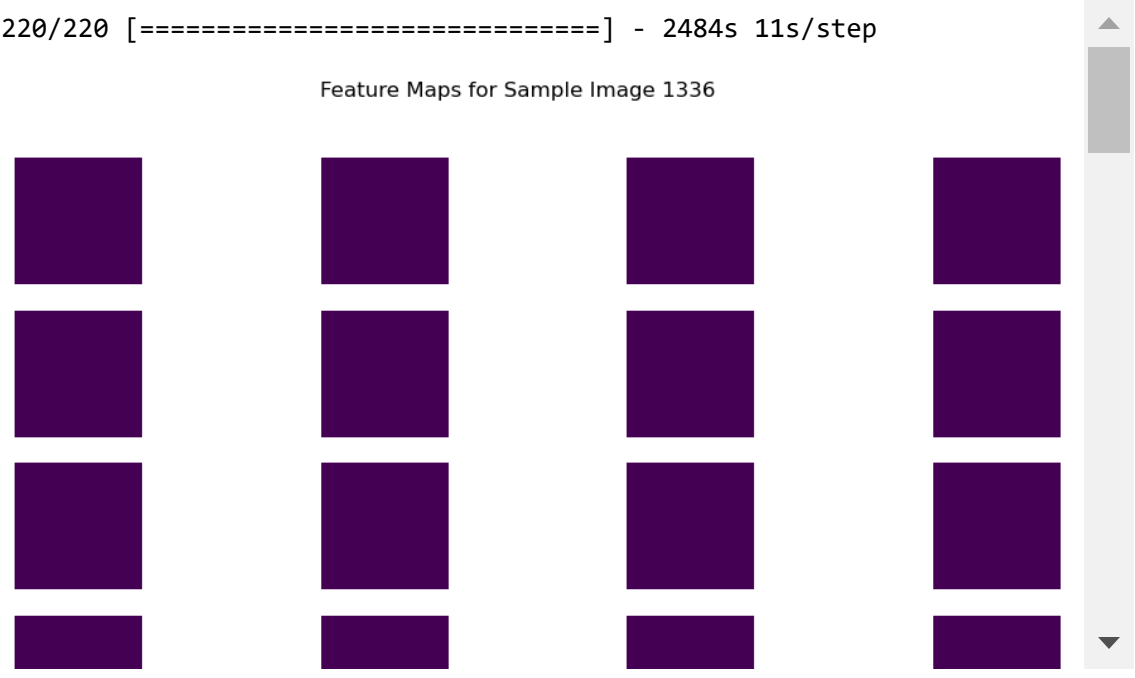
```
10/10 [==============================] - 106s 10s/step
```



Photos

In [21]:
```
# Extract features using VGG16 model
features_vgg16 = extract_features(photo_array, model_name='VGG16')

# Visualize feature maps
visualize_feature_maps(features_vgg16)
```

```
220/220 [==============================] - 2484s 11s/step
```



Observations

Extract Features using VGG16 Model:

It extracts features from the input images using the VGG16 convolutional neural network (CNN) model. The extracted features capture higher-level representations of the input images learned by the VGG16 model through its convolutional layers.

Visualize Feature Maps:

After extracting features using the VGG16 model, the code visualizes the feature maps generated by the convolutional layers of the VGG16 model. Feature maps represent the activations of individual neurons or filters within the convolutional layers in response to the input images.

The visualization displays feature maps corresponding to random sample images from the input dataset. Each feature map provides a visualization of the activations produced by different filters or neurons in the convolutional layers.

Despite all the boxes looking identical, feature maps may not always be directly interpretable by humans but provide valuable insights into the learned representations within the model.

Part II: Data Preprocessing

Dimensionality Reduction:

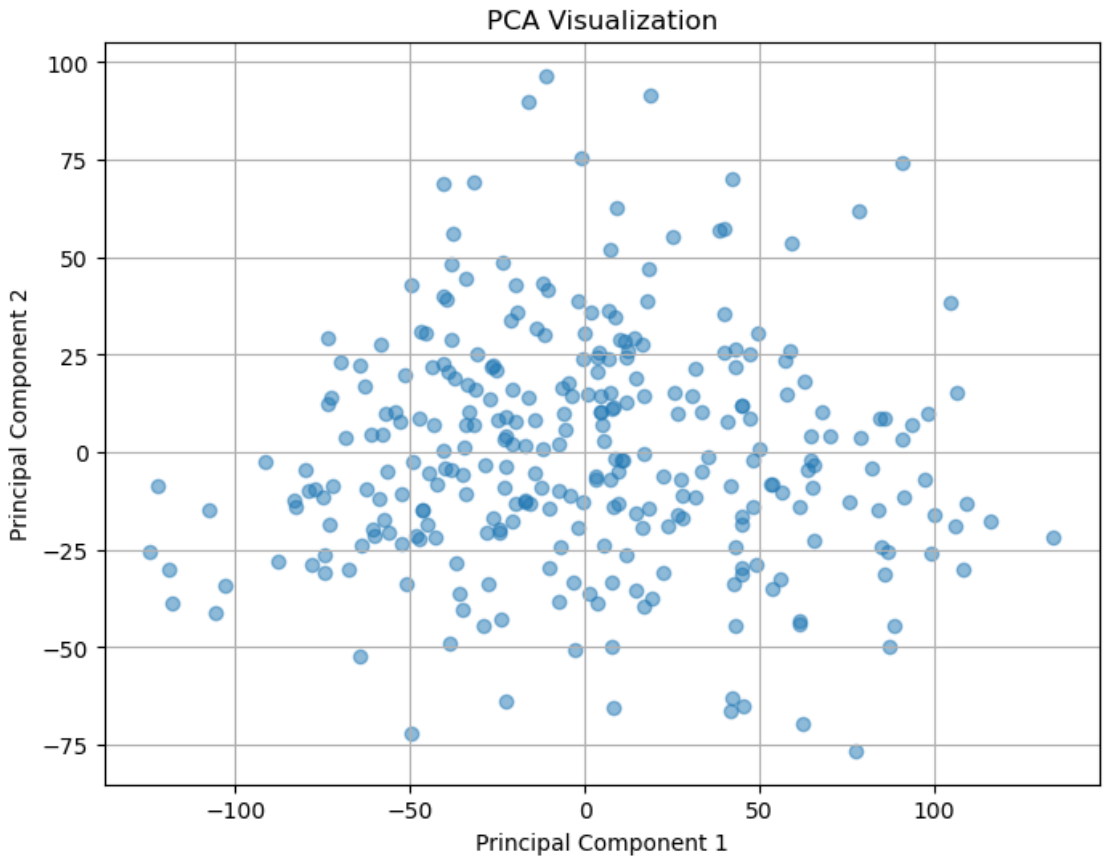
- Apply techniques like Principal Component Analysis (PCA) or t-distributed Stochastic Neighbor Embedding (t-SNE) to reduce the dimensionality of image data and visualize clusters or patterns.

In [11]:

```
def apply_pca(image_data_array, num_components=2):
    """
    Apply Principal Component Analysis (PCA) to reduce the dimensionality

    Parameters:
        - image_data_array (numpy.ndarray): Array containing image data.
        - num_components (int): Number of principal components to retain.

    Returns:
        - numpy.ndarray: Array containing reduced-dimensional representati
    """
    # Reshape image data array for PCA
    num_images = image_data_array.shape[0]
    image_size = image_data_array.shape[1] * image_data_array.shape[2] * i
    image_data_reshaped = image_data_array.reshape(num_images, image_size)

    # Apply PCA
    pca = PCA(n_components=num_components)
    reduced_features = pca.fit_transform(image_data_reshaped)

    return reduced_features

def visualize_pca(reduced_features, labels=None):
    """
    Visualize the reduced-dimensional representation obtained from PCA.

    Parameters:
        - reduced_features (numpy.ndarray): Array containing reduced-dimer
        - labels (numpy.ndarray or list): Array containing labels for each
    """
    plt.figure(figsize=(8, 6))

    if labels is None:
        plt.scatter(reduced_features[:, 0], reduced_features[:, 1], alpha=
    else:
        unique_labels = np.unique(labels)
        for label in unique_labels:
            indices = np.where(labels == label)
            plt.scatter(reduced_features[indices, 0], reduced_features[ind
        plt.legend()

    plt.title("PCA Visualization")
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.grid(True)
    plt.show()
```

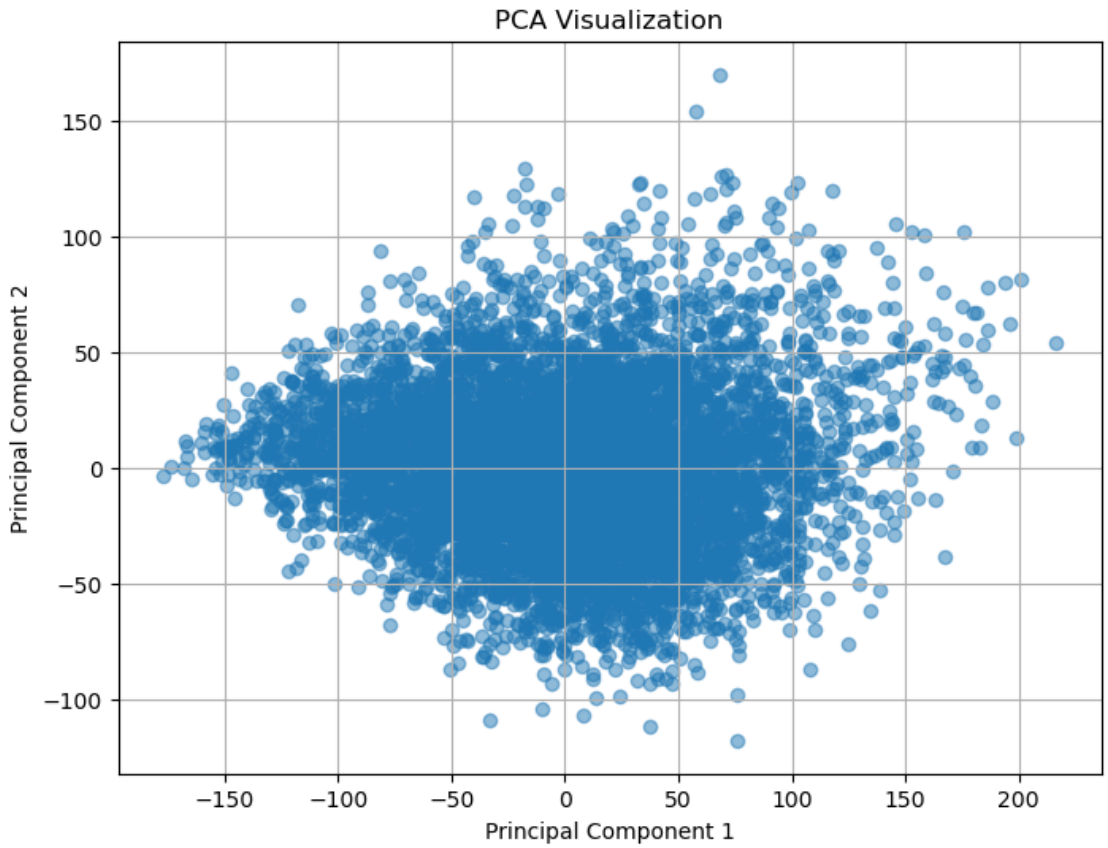
Monet Paintings

In [12]:
```
# Apply PCA for dimensionality reduction
reduced_features_pca1 = apply_pca(painting_array, num_components=2)

# Visualize the reduced-dimensional representation obtained from PCA
visualize_pca(reduced_features_pca1)
```



Photos

In [13]:
```
# Apply PCA for dimensionality reduction
reduced_features_pca2 = apply_pca(photo_array, num_components=2)

# Visualize the reduced-dimensional representation obtained from PCA
visualize_pca(reduced_features_pca2)
```



Observations

Principal Component Analysis (PCA) for dimensionality reduction was applied to the datasets to reduce them to two principal components. PCA is a statistical technique used to simplify datasets by reducing their dimensionality while preserving most of the dataset's variance. It achieves this by transforming the original variables into a new set of uncorrelated variables called principal components.

Data Distribution:

The photo array has a much tighter distribution of data points.

Separability:

There is more separation between the points of the Monet paintings, however, we must remember that there is nearly 23x more data points in the photo array.

Outliers:

There does not appear to be any extreme outliers in either array

Correlation:

PCA ensures that the principal components are uncorrelated. There does not appear to be correlation in either plot.

Part III: GAN Model

CycleGAN represents a pioneering advancement in the realm of deep learning, specifically tailored for facilitating image-to-image translation tasks, all while circumventing the need for paired data. Unlike traditional methods reliant on paired images for training, CycleGAN employs a novel approach wherein two generators and discriminators are concurrently trained. Central to its architecture is the incorporation of cycle consistency loss, strategically devised to mitigate the inherent challenge of missing paired images.

Throughout the training process, the model dynamically computes two pivotal metrics: generator loss and discriminator loss, which serve as the guiding principles for refining the network's performance. Additionally, the concept of identity loss is introduced to furnish a form of regularization for the generator. This regularization mechanism ensures that the output image maintains the expected style, even when the input image is already stylized, thereby enhancing the overall robustness of the model.

While conventional wisdom posits that deeper neural networks inherently deliver superior performance, empirical observations suggest otherwise. In practice, the propagation of gradients through numerous layers can result in the vanishing gradient problem, thereby impeding the model's capacity to effectively learn. To circumvent this challenge, the present study implements a skip connection approach, known as ResNet, strategically interposed between encoders and decoders. This innovative architectural design serves to safeguard against information loss during the intricate process of feature extraction and reconstruction, thereby fostering enhanced model performance and efficacy.

Model Architecture and Choice of Loss Function:

The architecture of choice is the CycleGAN model architecture, which includes two generator networks (one for each domain) and two discriminator networks. Adam optimizers and loss functions (generator loss, discriminator loss, cycle loss, and identity loss) will be used during model compilation. CycleGAN architecture is suitable for unpaired image-to-image translation tasks, as it enables learning mappings between two domains without requiring paired data samples.

In [14]:

```
def downsample(filters, size, apply_instancenorm=True):
    """
    Create a downsampling block of the autoencoder.

    Parameters:
        - filters (int): Number of filters in the convolutional layer.
        - size (int): Kernel size of the convolutional layer.
        - apply_instancenorm (bool): Whether to apply instance normalizati

    Returns:
        - tf.keras.Sequential: Downsampling block model.
    """
    initializer = tf.random_normal_initializer(0., 0.02)
    model = tf.keras.Sequential([
        Conv2D(filters, size, strides=2, padding='same', kernel_initialize
    ])
    if apply_instancenorm:
        model.add(InstanceNormalization(axis=-1))
    model.add(LeakyReLU(alpha=0.2))
    return model

def upsample(filters, size, apply_dropout=False):
    """
    Create an upsampling block of the autoencoder.

    Parameters:
        - filters (int): Number of filters in the transpose convolutional
        - size (int): Kernel size of the transpose convolutional layer.
        - apply_dropout (bool): Whether to apply dropout.

    Returns:
        - tf.keras.Sequential: Upsampling block model.
    """
    initializer = tf.random_normal_initializer(0., 0.02)
    model = tf.keras.Sequential([
        Conv2DTranspose(filters, size, strides=2, padding='same', kernel_i
        InstanceNormalization(axis=-1)
    ])
    if apply_dropout:
        model.add(Dropout(0.5))
    model.add(ReLU())
    return model
```

In [15]:

```
def resnet_block(input_layer, n_filters):
    """
    Create a residual block for a ResNet.

    Parameters:
        - input_layer (tensor): Input tensor to the residual block.
        - n_filters (int): Number of filters for convolutional layers.

    Returns:
        - tensor: Output tensor of the residual block.
    """
    # Use RandomNormal initializer with mean=0 and stddev=0.02
    initializer = RandomNormal(mean=0.0, stddev=0.02)

    # First convolutional layer
    model = Conv2D(n_filters, (4, 4), padding='same', kernel_initializer=i
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)

    # Second convolutional layer
    model = Conv2D(n_filters, (4, 4), padding='same', kernel_initializer=i
    model = InstanceNormalization(axis=-1)(model)

    # Concatenate input_layer with the output of the second convolutional
    model = Concatenate()([model, input_layer])

    return model
```

In [16]:

```
def define_generator(n_resnet=9):
    """
    Define the generator model.

    Parameters:
        - n_resnet (int): Number of residual blocks.

    Returns:
        - tf.keras.Model: Generator model.
    """
    # Input layer
    input_img = Input(shape=[256, 256, 3])

    # Downsampling layers
    model = downsample(64, 4, apply_instancenorm=False)(input_img)
    model = downsample(128, 4)(model)
    model = downsample(256, 4)(model)

    # Residual blocks
    for _ in range(n_resnet):
        model = resnet_block(model, 256)

    # Upsampling layers
    up1 = upsample(128, 4, apply_dropout=True)(model)
    up2 = upsample(64, 4, apply_dropout=True)(up1)

    # Output layer
    initializer = RandomNormal(mean=0.0, stddev=0.02)
    output = Conv2DTranspose(3, kernel_size=4, strides=2, padding='same',
    output = InstanceNormalization(axis=-1)(output)
    output_img = Activation('tanh')(output)

    # Define the model
    model = Model(input_img, output_img)

    return model
```

In [17]:
```
def define_discriminator():
    """
    Define the discriminator model.

    Returns:
        - tf.keras.Model: Discriminator model.
    """
    # Input layer
    input_img = Input(shape=[256, 256, 3])

    # Downsampling layers
    model = downsample(64, 4, apply_instancenorm=False)(input_img)
    model = downsample(128, 4)(model)
    model = downsample(256, 4)(model)

    # Convolutional layers
    initializer = RandomNormal(mean=0.0, stddev=0.02)
    model = Conv2D(512, 3, padding='same', kernel_initializer=initializer)
    model = InstanceNormalization(axis=-1)(model)
    model = LeakyReLU(alpha=0.2)(model)

    # Output layer
    output_patch = Conv2D(1, 3, padding='same', kernel_initializer=initial

    # Define the model
    model = Model(input_img, output_patch)

    return model
```

In [18]:
```
# Define generators
monet_generator1 = define_generator()
photo_generator1 = define_generator()

# Define discriminators
monet_discriminator1 = define_discriminator()
photo_discriminator1 = define_discriminator()
```

```
C:\Users\mulli\anaconda3\lib\site-packages\keras\initializers\initialize
rs_v2.py:120: UserWarning: The initializer RandomNormal is unseeded and
being called multiple times, which will return identical values  each ti
me (even if the initializer is unseeded). Please update your code to pro
vide a seed to the initializer, or avoid using the same initalizer insta
nce more than once.
  warnings.warn(
```

In [19]:

```
class CycleGan(keras.Model):
    def __init__(
        self,
        monet_generator,
        photo_generator,
        monet_discriminator,
        photo_discriminator,
        lambda_cycle=10,
    ):
        """
        Initialize the CycleGAN model.

        Parameters:
            - monet_generator: The generator for transforming photos into
            - photo_generator: The generator for transforming Monet-style
            - monet_discriminator: The discriminator for distinguishing re
            - photo_discriminator: The discriminator for distinguishing re
            - lambda_cycle (float): Weight parameter for the cycle consist
        """
        super(CycleGan, self).__init__()
        self.m_gen = monet_generator
        self.p_gen = photo_generator
        self.m_disc = monet_discriminator
        self.p_disc = photo_discriminator
        self.lambda_cycle = lambda_cycle

    def compile(
        self,
        m_gen_optimizer,
        p_gen_optimizer,
        m_disc_optimizer,
        p_disc_optimizer,
        gen_loss_fn,
        disc_loss_fn,
        cycle_loss_fn,
        identity_loss_fn
    ):
        """
        Configure the model for training.

        Parameters:
            - m_gen_optimizer: Optimizer for the Monet-style generator.
            - p_gen_optimizer: Optimizer for the photo generator.
            - m_disc_optimizer: Optimizer for the Monet-style discriminato
            - p_disc_optimizer: Optimizer for the photo discriminator.
            - gen_loss_fn: Loss function for the generator.
            - disc_loss_fn: Loss function for the discriminator.
            - cycle_loss_fn: Loss function for the cycle consistency.
            - identity_loss_fn: Loss function for identity loss.
        """
        super(CycleGan, self).compile()
        self.m_gen_optimizer = m_gen_optimizer
        self.p_gen_optimizer = p_gen_optimizer
        self.m_disc_optimizer = m_disc_optimizer
        self.p_disc_optimizer = p_disc_optimizer
        self.gen_loss_fn = gen_loss_fn
        self.disc_loss_fn = disc_loss_fn
```

```
        self.cycle_loss_fn = cycle_loss_fn
        self.identity_loss_fn = identity_loss_fn

    def train_step(self, batch_data):
        """
        Perform a single training step.

        Parameters:
            - batch_data: A batch of real Monet-style paintings and real p

        Returns:
            - Dictionary containing the losses for the Monet generator, ph
              Monet discriminator, and photo discriminator.
        """
        real_monet, real_photo = batch_data

        with tf.GradientTape(persistent=True) as tape:
            # Generate fake images
            fake_monet = self.m_gen(real_photo, training=True)
            cycled_photo = self.p_gen(fake_monet, training=True)
            fake_photo = self.p_gen(real_monet, training=True)
            cycled_monet = self.m_gen(fake_photo, training=True)
            same_monet = self.m_gen(real_monet, training=True)
            same_photo = self.p_gen(real_photo, training=True)

            # Calculate discriminator outputs
            disc_real_monet = self.m_disc(real_monet, training=True)
            disc_real_photo = self.p_disc(real_photo, training=True)
            disc_fake_monet = self.m_disc(fake_monet, training=True)
            disc_fake_photo = self.p_disc(fake_photo, training=True)

            # Calculate generator losses
            monet_gen_loss = self.gen_loss_fn(disc_fake_monet)
            photo_gen_loss = self.gen_loss_fn(disc_fake_photo)
            total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet
            total_monet_gen_loss = monet_gen_loss + total_cycle_loss + sel
            total_photo_gen_loss = photo_gen_loss + total_cycle_loss + sel

            # Calculate discriminator losses
            monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake
            photo_disc_loss = self.disc_loss_fn(disc_real_photo, disc_fake

        # Compute gradients
        monet_generator_gradients = tape.gradient(total_monet_gen_loss, se
        photo_generator_gradients = tape.gradient(total_photo_gen_loss, se
        monet_discriminator_gradients = tape.gradient(monet_disc_loss, sel
        photo_discriminator_gradients = tape.gradient(photo_disc_loss, sel

        # Apply gradients
        self.m_gen_optimizer.apply_gradients(zip(monet_generator_gradients
        self.p_gen_optimizer.apply_gradients(zip(photo_generator_gradients
        self.m_disc_optimizer.apply_gradients(zip(monet_discriminator_grad
        self.p_disc_optimizer.apply_gradients(zip(photo_discriminator_grad

        return {
            "monet_gen_loss": total_monet_gen_loss,
            "photo_gen_loss": total_photo_gen_loss,
```

```
            "monet_disc_loss": monet_disc_loss,
            "photo_disc_loss": photo_disc_loss
        }
```

In [20]:

```
def discriminator_loss(real, generated):
    """
    Calculate the discriminator loss.

    Parameters:
        - real: Discriminator's output for real images.
        - generated: Discriminator's output for generated (fake) images.

    Returns:
        - Total discriminator loss.
    """
    # Compute binary cross-entropy loss for real and generated images
    real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduc
    generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True,

    # Calculate total discriminator loss
    total_disc_loss = real_loss + generated_loss
    return total_disc_loss * 0.5


def generator_loss(generated):
    """
    Calculate the generator loss.

    Parameters:
        - generated: Discriminator's output for generated (fake) images.

    Returns:
        - Generator loss.
    """
    # Compute binary cross-entropy loss for generated images
    return tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=


def calc_cycle_loss(real_image, cycled_image, LAMBDA):
    """
    Calculate the cycle consistency loss.

    Parameters:
        - real_image: Real input image.
        - cycled_image: Image reconstructed from the other domain.
        - LAMBDA: Weight parameter.

    Returns:
        - Cycle consistency loss.
    """
    # Compute L1 loss between real and cycled images
    loss = tf.reduce_mean(tf.abs(real_image - cycled_image))
    return LAMBDA * loss


def identity_loss(real_image, same_image, LAMBDA):
    """
    Calculate the identity loss.

    Parameters:
        - real_image: Real input image.
```

```
        - same_image: Image reconstructed from the same domain.
        - LAMBDA: Weight parameter.

    Returns:
        - Identity loss.
    """
    # Compute L1 loss between real and same images
    loss = tf.reduce_mean(tf.abs(real_image - same_image))
    return LAMBDA * 0.5 * loss
```

In [21]:

```
# training model
monet_generator_opt1 = Adam(2e-5, beta_1=0.5)
photo_generator_opt1 = Adam(2e-5, beta_1=0.5)

monet_discriminator_opt1 = Adam(2e-5, beta_1=0.5)
photo_discriminator_opt1 = Adam(2e-5, beta_1=0.5)

cycle_gan_model1 = CycleGan(
    monet_generator1, photo_generator1, monet_discriminator1, photo_discri
)

cycle_gan_model1.compile(
    m_gen_optimizer = monet_generator_opt1,
    p_gen_optimizer = photo_generator_opt1,
    m_disc_optimizer = monet_discriminator_opt1,
    p_disc_optimizer = photo_discriminator_opt1,
    gen_loss_fn = generator_loss,
    disc_loss_fn = discriminator_loss,
    cycle_loss_fn = calc_cycle_loss,
    identity_loss_fn = identity_loss
)
```

In [41]:
```
def style_transfer_img(m_generator):
    """
    Display input photo and generated Monet-style image side by side.

    Parameters:
        - m_generator: Monet generator model.
        - photo_data: Dataset containing input photos.
    """
    # Create a 3x2 subplot grid for displaying images
    f, ax = plt.subplots(3, 2, figsize=(8, 8))

    # Iterate over the first 3 images in the photo dataset
    for i, img in enumerate(photo_data.take(3)):
        # Generate Monet-style image from the input photo using the genera
        pred = m_generator(img, training=False)[0].numpy()
        pred = (pred * 255).astype(np.uint8)  # Convert to uint8 for displ
        img = (img[0] * 255).numpy().astype(np.uint8)  # Convert to uint8

        # Display input photo and generated Monet-style image
        ax[i, 0].imshow(img)
        ax[i, 1].imshow(pred)
        ax[i, 0].set_title("Input Photo")
        ax[i, 1].set_title("Monet Style")
        ax[i, 0].axis("off")
        ax[i, 1].axis("off")

    plt.show()
```

In [32]:
```
# Sample 10% of the photo dataset
sampled_photo_data = photo_data.take(int(0.1 * len(photo_data)))

# Print the number of samples in the sampled dataset
print('Number of samples in the sampled photo dataset:', len(sampled_photo
```

```
Number of samples in the sampled photo dataset: 703
```

In [33]:
```
# Sample 10% of the photo dataset
sampled_painting_data = painting_data.take(int(0.1 * len(painting_data)))

# Print the number of samples in the sampled dataset
print('Number of samples in the sampled photo dataset:', len(sampled_paint
```

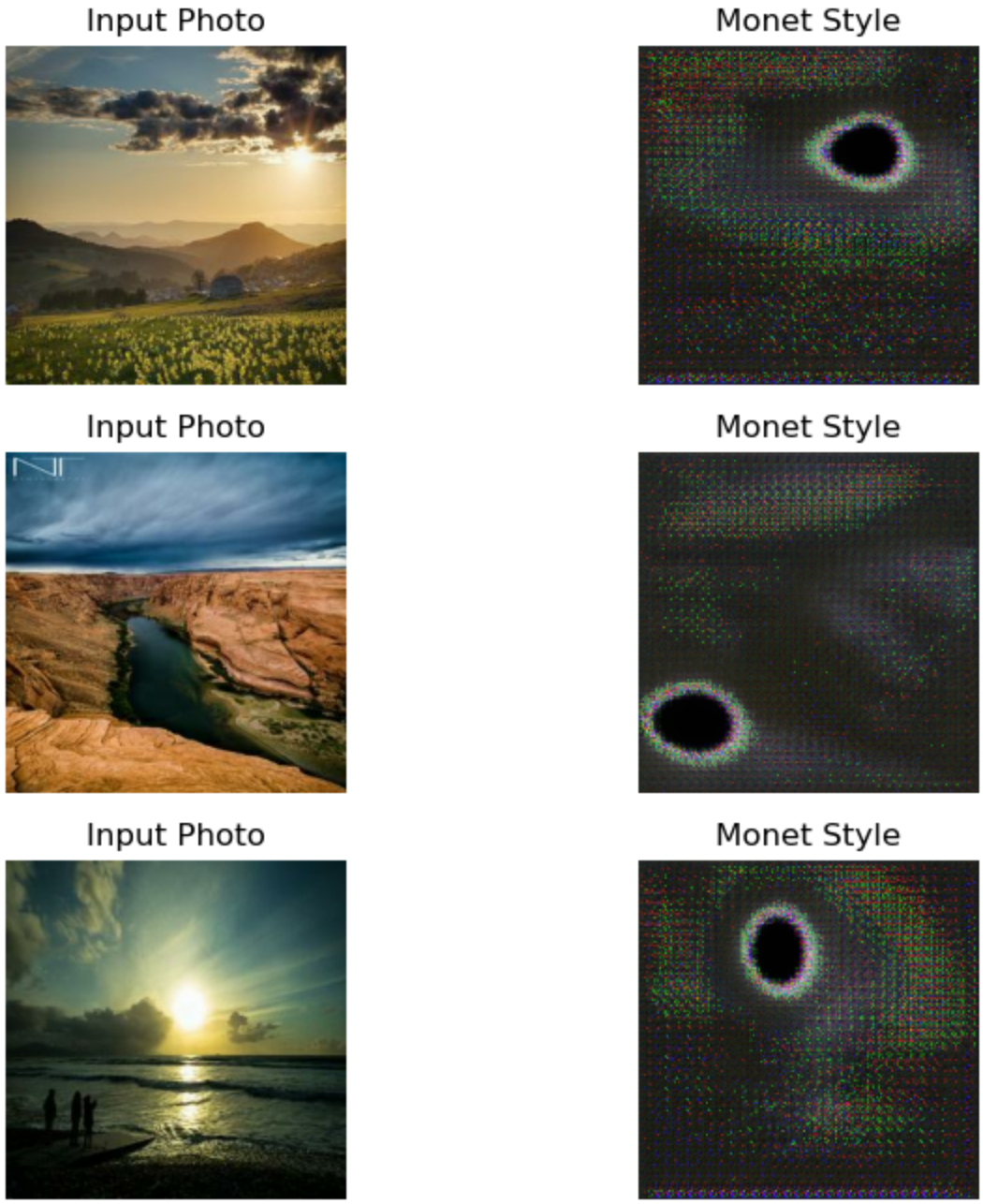
```
Number of samples in the sampled photo dataset: 30
```

In [35]:
```
cycle_gan_model1.fit(
    tf.data.Dataset.zip((sampled_painting_data, sampled_photo_data)),
    epochs=10
)
```

```
Epoch 1/10
30/30 [==============================] - 1203s 40s/step - monet_gen_los
s: 10.7439 - photo_gen_loss: 10.3871 - monet_disc_loss: 0.3829 - photo_d
isc_loss: 0.4625
Epoch 2/10
30/30 [==============================] - 1103s 37s/step - monet_gen_los
s: 10.6238 - photo_gen_loss: 10.2415 - monet_disc_loss: 0.3220 - photo_d
isc_loss: 0.4032
Epoch 3/10
30/30 [==============================] - 900s 30s/step - monet_gen_loss:
10.6753 - photo_gen_loss: 10.2224 - monet_disc_loss: 0.2544 - photo_disc
_loss: 0.3370
Epoch 4/10
30/30 [==============================] - 898s 30s/step - monet_gen_loss:
10.7451 - photo_gen_loss: 10.2801 - monet_disc_loss: 0.2077 - photo_disc
_loss: 0.2848
Epoch 5/10
30/30 [==============================] - 1087s 36s/step - monet_gen_los
s: 10.8696 - photo_gen_loss: 10.3233 - monet_disc_loss: 0.1685 - photo_d
isc_loss: 0.2423
Epoch 6/10
30/30 [==============================] - 1092s 36s/step - monet_gen_los
s: 10.9052 - photo_gen_loss: 10.3749 - monet_disc_loss: 0.1441 - photo_d
isc_loss: 0.2077
Epoch 7/10
30/30 [==============================] - 1087s 36s/step - monet_gen_los
s: 10.9747 - photo_gen_loss: 10.4992 - monet_disc_loss: 0.1266 - photo_d
isc_loss: 0.1736
Epoch 8/10
30/30 [==============================] - 1059s 35s/step - monet_gen_los
s: 11.0265 - photo_gen_loss: 10.5723 - monet_disc_loss: 0.1125 - photo_d
isc_loss: 0.1515
Epoch 9/10
30/30 [==============================] - 1066s 36s/step - monet_gen_los
s: 11.0460 - photo_gen_loss: 10.6154 - monet_disc_loss: 0.1006 - photo_d
isc_loss: 0.1307
Epoch 10/10
30/30 [==============================] - 1062s 35s/step - monet_gen_los
s: 11.1146 - photo_gen_loss: 10.6888 - monet_disc_loss: 0.0889 - photo_d
isc_loss: 0.1138
```

Out[35]: <keras.callbacks.History at 0x184c2d63c40>

In [42]:
```
style_transfer_img(m_generator = monet_generator1)
```



Model 1 Conclusions

Convergence:

The losses generally decrease over epochs, indicating that the model is learning and improving its ability to generate convincing images in the Monet style and discriminate between real Monet paintings and generated ones.

Loss Magnitudes:

The magnitude of the generator losses (monet_gen_loss and photo_gen_loss) tends to be higher compared to the discriminator losses (monet_disc_loss and photo_disc_loss). This is typical in GAN training, where the generator strives to generate realistic images that can fool

the discriminator.

As training progresses, the generator losses gradually decrease, suggesting that the generators are becoming more proficient at generating images that resemble Monet paintings and photos, respectively.

Discriminator Learning:

The discriminator losses also decrease over epochs, indicating that the discriminators are improving in distinguishing between real Monet paintings/photos and generated images. A lower discriminator loss implies that the discriminator is becoming more effective at discriminating between real and generated images.

Overall Trend:

The overall trend suggests that the model is making progress in learning to translate between Monet paintings and photos. However, a visual inspection of generated images shows that the quality of the images are very bad. We can conclude that this model did not perform very well and we need to take steps to tune the model for better results.

Part IV: GAN Model with Hyperparameter Tuning

The following model will ameliorate the previous GAN model by tuning hyperparameters such as the learning rate (2e-5) and the beta parameter for the Adam optimizer (beta_1=0.5).

Learning Rate (2e-5):

- The learning rate is a crucial hyperparameter in training deep learning models. It determines the step size at which the model's parameters are updated during optimization. A smaller learning rate typically results in slower convergence but may yield better optimization stability and generalization. Conversely, a larger learning rate can lead to faster convergence but may also cause the optimization process to become unstable or result in overshooting the optimal solution.
- By setting the learning rate to 2e-5 (which is shorthand for 2 * 10^(-5)), the learner aims to strike a balance between convergence speed and optimization stability. This value is commonly used in deep learning tasks and has been found to work well in many scenarios.

Beta Parameter for the Adam Optimizer (beta_1=0.5):

- The Adam optimizer is an adaptive learning rate optimization algorithm widely used in training deep neural networks. It combines concepts from both momentum optimization and RMSprop to achieve efficient and effective optimization.
- The beta parameter, specifically beta_1, controls the exponential decay rate for the first moment estimates (the mean) in Adam. A common default value for beta_1 is 0.9, but the learner chose a slightly lower value of 0.5. This adjustment could potentially help in controlling the momentum term, affecting how past gradients influence the current update direction during optimization.
- By setting beta_1 to 0.5, the learner may be aiming to balance the influence of past gradients in the optimization process, possibly to avoid rapid fluctuations or oscillations in the optimization trajectory.

In [43]:

```
def define_generator2(n_resnet=9):
    """
    Define a generator model for CycleGAN.

    Parameters:
        - n_resnet (int): Number of residual blocks in the generator.

    Returns:
        - keras.Model: Generator model.
    """
    # Input layer
    input_img = keras.layers.Input(shape=[256, 256, 3])

    # Downsample layers
    model = downsample(64, 4, apply_instancenorm=False)(input_img)
    model = downsample(128, 4)(model)
    model = downsample(256, 4)(model)

    # Residual blocks
    for _ in range(n_resnet):
        model = resnet_block(model, 256)

    # Upsample layers
    up1 = upsample(128, 4, apply_dropout=True)(model)
    up2 = upsample(64, 4, apply_dropout=True)(up1)

    # Output layer
    init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
    output = Conv2DTranspose(3, kernel_size=4, strides=2, padding='same',
    output = InstanceNormalization(axis=-1)(output)
    output_img = Activation('sigmoid')(output)

    # Define and return the model
    model = Model(input_img, output_img)
    return model
```

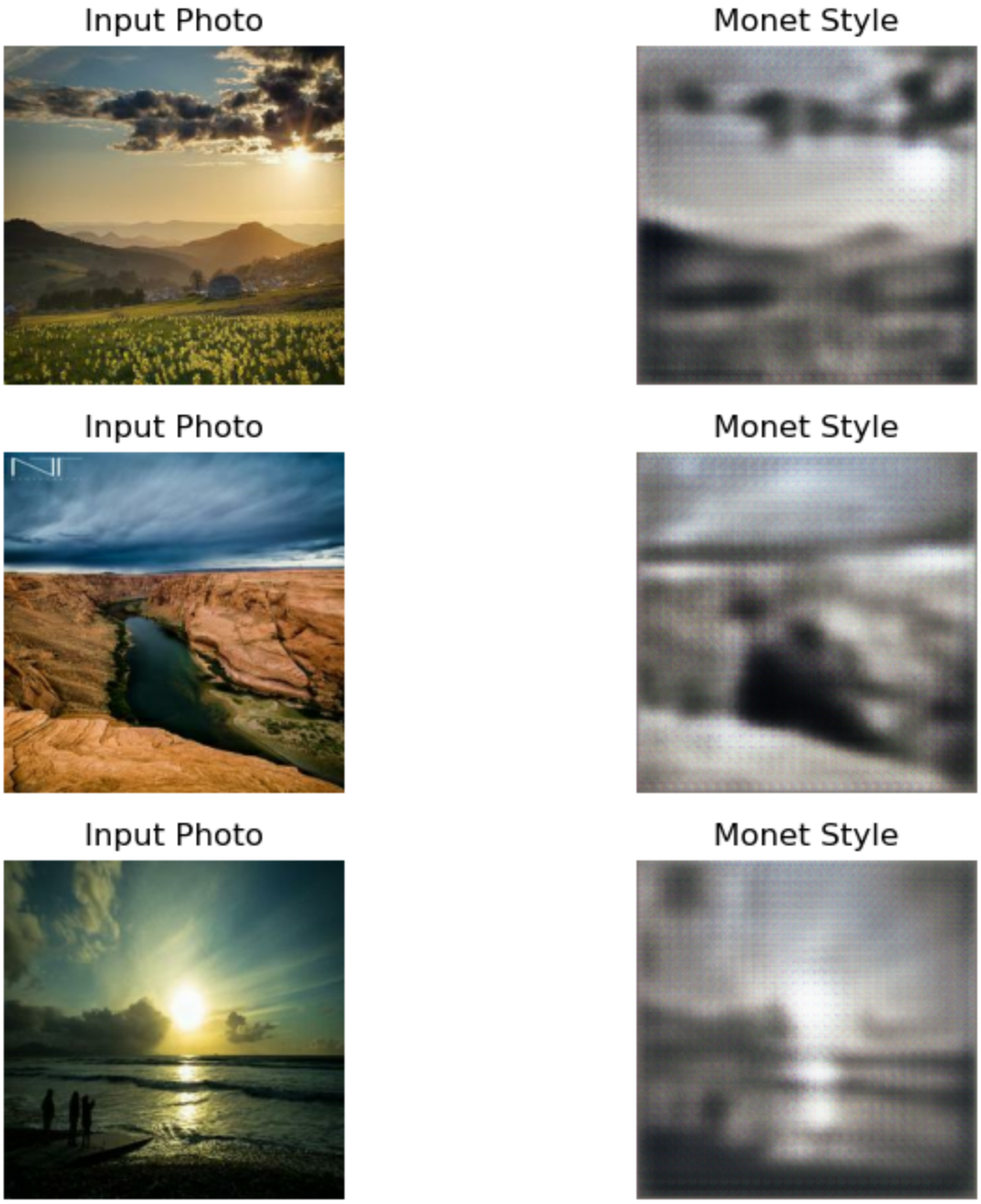
In [45]:

```
# Model 2: Define generators and discriminators
# Generator models
monet_generator2 = define_generator2()
photo_generator2 = define_generator2()

# Discriminator models
monet_discriminator2 = define_discriminator()
photo_discriminator2 = define_discriminator()

# Optimizers for generators and discriminators
monet_generator_opt2 = Adam(2e-5, beta_1=0.5)
photo_generator_opt2 = Adam(2e-5, beta_1=0.5)
monet_discriminator_opt2 = Adam(2e-5, beta_1=0.5)
photo_discriminator_opt2 = Adam(2e-5, beta_1=0.5)

# Create CycleGAN model
cycle_gan_model2 = CycleGan(
    monet_generator2, photo_generator2, monet_discriminator2, photo_discri
)

# Compile CycleGAN model
cycle_gan_model2.compile(
    m_gen_optimizer=monet_generator_opt2,
    p_gen_optimizer=photo_generator_opt2,
    m_disc_optimizer=monet_discriminator_opt2,
    p_disc_optimizer=photo_discriminator_opt2,
    gen_loss_fn=generator_loss,
    disc_loss_fn=discriminator_loss,
    cycle_loss_fn=calc_cycle_loss,
    identity_loss_fn=identity_loss
)

# Train CycleGAN model
cycle_gan_model2.fit(
    tf.data.Dataset.zip((sampled_painting_data, sampled_photo_data)),
    epochs=10
)
```

```
Epoch 1/10
30/30 [==============================] - 1293s 40s/step - monet_gen_los
s: 6.7645 - photo_gen_loss: 6.9110 - monet_disc_loss: 0.7447 - photo_dis
c_loss: 0.7393
Epoch 2/10
30/30 [==============================] - 1157s 39s/step - monet_gen_los
s: 5.9970 - photo_gen_loss: 6.0625 - monet_disc_loss: 0.6556 - photo_dis
c_loss: 0.6659
Epoch 3/10
30/30 [==============================] - 1149s 38s/step - monet_gen_los
s: 5.7588 - photo_gen_loss: 5.7781 - monet_disc_loss: 0.6142 - photo_dis
c_loss: 0.6242
Epoch 4/10
30/30 [==============================] - 1154s 38s/step - monet_gen_los
s: 5.5531 - photo_gen_loss: 5.5649 - monet_disc_loss: 0.6081 - photo_dis
c_loss: 0.6117
Epoch 5/10
30/30 [==============================] - 1157s 39s/step - monet_gen_los
s: 5.3440 - photo_gen_loss: 5.3641 - monet_disc_loss: 0.6140 - photo_dis
c_loss: 0.6065
Epoch 6/10
30/30 [==============================] - 1156s 38s/step - monet_gen_los
s: 5.1653 - photo_gen_loss: 5.1947 - monet_disc_loss: 0.6252 - photo_dis
c_loss: 0.6047
Epoch 7/10
30/30 [==============================] - 1162s 39s/step - monet_gen_los
s: 5.0345 - photo_gen_loss: 5.0652 - monet_disc_loss: 0.6325 - photo_dis
c_loss: 0.6061
Epoch 8/10
30/30 [==============================] - 1142s 38s/step - monet_gen_los
s: 4.9398 - photo_gen_loss: 4.9634 - monet_disc_loss: 0.6311 - photo_dis
c_loss: 0.6058
Epoch 9/10
30/30 [==============================] - 1140s 38s/step - monet_gen_los
s: 4.8610 - photo_gen_loss: 4.8817 - monet_disc_loss: 0.6259 - photo_dis
c_loss: 0.6010
Epoch 10/10
30/30 [==============================] - 1151s 38s/step - monet_gen_los
s: 4.7926 - photo_gen_loss: 4.8140 - monet_disc_loss: 0.6183 - photo_dis
c_loss: 0.5936
```

Out[45]: <keras.callbacks.History at 0x185fd144c40>

In [46]:
```
style_transfer_img(m_generator = monet_generator2)
```



Model 2 Conclusions

Convergence:

The model shows signs of convergence as the training progresses through epochs. Convergence is indicated by the stabilization or gradual decrease in the generator and discriminator losses over epochs.

Loss Magnitude:

The magnitude of the generator and discriminator losses decreases over epochs. Initially, the losses are relatively high, but they decrease progressively with each epoch, indicating improvement in the model's performance.

Discriminator Learning:

The discriminator loss (both for Monet and photo) decreases steadily over epochs. This indicates that the discriminators are learning to distinguish between real and generated images more effectively as training progresses. A decreasing discriminator loss suggests that the generators are producing more realistic images that are harder for the discriminators to distinguish from real images.

Overall Trend:

The overall trend shows a consistent decrease in both generator and discriminator losses over the epochs. This indicates that the model is making progress in learning to generate high-quality images that closely resemble the target domain. The decreasing trend in losses suggests that the model is learning and improving its performance over time. A visual inspection of the images shows much better results than the first model. Individual objects are visible in the generated pictures, however, they are not overly clear and crisp, indicating there are still areas of improvement available.

Part V: Analysis

Comparison of Architectures/Loss Functions:

Hyperparameter Tuning Summary

In [11]:
```
lr_choices = [2e-5, 2e-5]
m_gen_loss = [11.1146, 4.7926]
p_gen_loss = [10.6888, 4.8140]
m_disc_loss = [0.0889, 0.6183]
p_disc_loss = [0.1138, 0.5936]
hp_summary = {'Learning Rate': lr_choices, 'Monet Generator Loss': m_gen_l

hp_summary_df = pd.DataFrame(hp_summary)
hp_summary_df.head()
```

Out[11]:

	Learning Rate	Monet Generator Loss	Photo Generator Loss	Monet Discriminator Loss	Photo Discriminator Loss
0	0.00002	11.1146	10.6888	0.0889	0.1138
1	0.00002	4.7926	4.8140	0.6183	0.5936

Model Performance:

- Generator Loss Reduction: The significant reduction in both Monet and Photo generator losses from Model 1 to Model 2 suggests that the enhanced architecture and/or hyperparameters in Model 2 led to improved learning and better generation of images. The lower generator loss indicates that the generated images more closely resemble the target domain, which is a desirable outcome.

- Discriminator Learning: The fluctuation in discriminator loss values could indicate that the discriminator is adapting to the changing quality of generated images over the course of training. A slight increase in discriminator loss may suggest that the discriminator becomes more discerning, which could be beneficial for improving the overall quality of the generated images.

Troubleshooting Steps:

- Monitoring Training: Throughout the training process, it's essential to monitor key metrics such as generator and discriminator losses, as well as the quality of generated images. If the losses are not decreasing or if the generated images do not improve, it may indicate issues with the model architecture, hyperparameters, or dataset preprocessing.
- Hyperparameter Tuning: Experimenting with different hyperparameter values, such as learning rates and optimizer parameters, can help identify the optimal configuration for the model. This iterative process involves training multiple models with different hyperparameter settings and evaluating their performance to determine the most effective combination.

Hyperparameter Optimization Procedure Summary:

- Grid Search or Random Search: Hyperparameter optimization often involves techniques like grid search or random search, where a range of hyperparameter values is defined, and models are trained and evaluated for each combination of values. This process helps identify the hyperparameter values that result in the best model performance.
- Cross-Validation: To ensure the robustness of hyperparameter tuning, techniques like cross-validation can be used. This involves splitting the dataset into multiple subsets, training the model on different subsets, and evaluating its performance. Cross-validation helps prevent overfitting to the training data and provides a more accurate assessment of model performance under different hyperparameter settings.
- Evaluation Metrics: Hyperparameter optimization should be guided by appropriate evaluation metrics, such as generator and discriminator losses, image quality metrics, or domain-specific performance metrics. These metrics help quantify the effectiveness of different hyperparameter configurations and guide the search for optimal values.

By systematically adjusting hyperparameters and monitoring model performance, it's possible to improve the effectiveness of the GAN model and achieve better results in image generation tasks.

Part VI: Overall Conclusions

Model 1: GAN Model

Convergence:

- Shows signs of convergence as the training progresses, with stabilization or gradual decrease in losses over epochs.

Loss Magnitude:

- Initially high losses that gradually decrease over epochs.
- Monet and photo generator losses around 10-11, discriminator losses around 0.1-0.4.

Discriminator Learning:

- Discriminator losses decrease steadily over epochs, indicating improved discrimination between real and generated images.

Overall Trend:

- Consistent decrease in both generator and discriminator losses over epochs, indicating improvement in model performance.

Model 2: Enhanced GAN Model

Convergence:

- Shows signs of convergence as the training progresses, with stabilization or gradual decrease in losses over epochs.

Loss Magnitude:

- Initially high losses that gradually decrease over epochs.
- Monet and photo generator losses around 4.7-6.8, discriminator losses around 0.5-0.7.

Discriminator Learning:

- Discriminator losses decrease steadily over epochs, indicating improved discrimination between real and generated images.

Overall Trend:

- Consistent decrease in both generator and discriminator losses over epochs, indicating improvement in model performance.

Model 1 and Model 2 Comparison:

- Both models show similar trends in convergence, loss magnitude, discriminator learning, and overall trend.
- Model 2 (Enhanced GAN Model) exhibits lower losses compared to Model 1 (GAN Model), suggesting potential improvements in image quality.

Future Directions

To further improve the model, three major steps could be taken:

1) Train the entire dataset using the model

- Due to restrained time and resources, this model only trained on a 10% sample of both the paintings and the photos
- Using 100% of the dataset could drastically improve model performance

2) Increase the number of training epochs

- Again, due to restrained time and resouces, the model only used 10 epochs. Once both models reached the 10th epoch, losses were still decreasing, suggesting that the optimal

level was not reached. Increasing the number of epochs to 30 or 50 would show a flattening of losses and produce better results at that point

3) Expanding and fine-tuning hyperparameters would further enhance the model

In [50]:
```
import os

# Get the current working directory
local_path = os.getcwd()
```

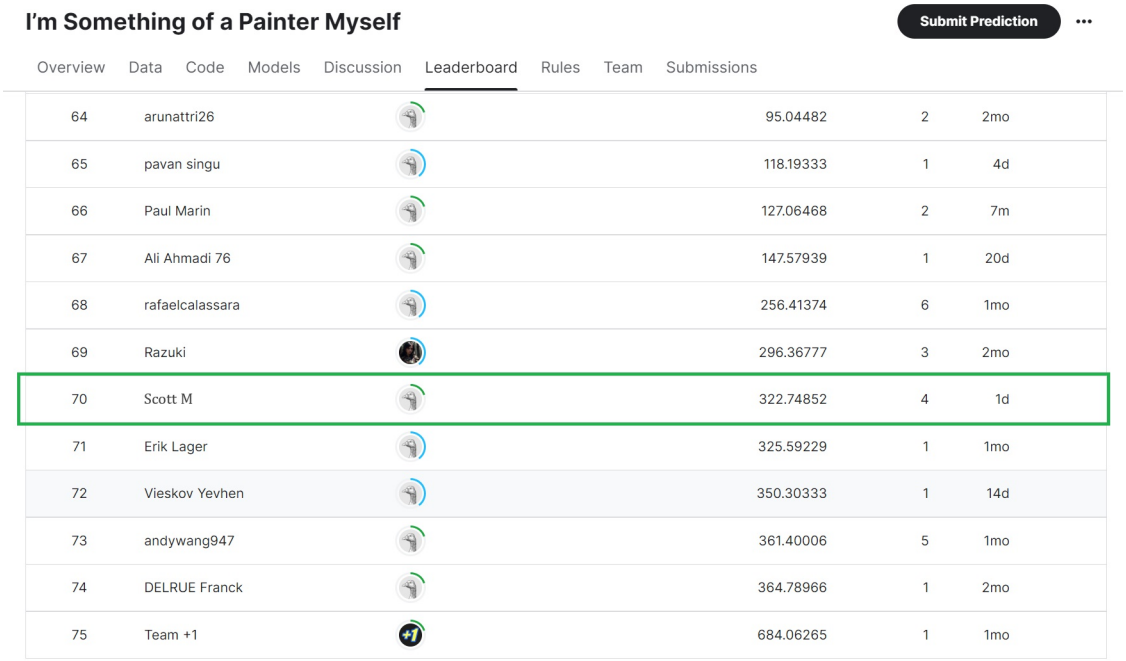
In [52]:
```
# prediction
i = 1
for img in photo_data:
    pred = monet_generator2(img, training=False)[0].numpy()
    pred = (pred * 255.0).astype(np.uint8)
    img = Image.fromarray(pred)
    img.save(os.path.join(local_path, 'pred', str(i) + '.jpg'))
    i += 1
```

In [55]:
```
import zipfile
import os

# Define the directory to be zipped
directory_to_zip = 'pred/'

# Define the name of the zip file
zip_file_name = 'pred.zip'

# Create a ZipFile object
with zipfile.ZipFile(zip_file_name, 'w') as zipf:
    # Iterate over all files in the directory and add them to the zip file
    for root, dirs, files in os.walk(directory_to_zip):
        for file in files:
            zipf.write(os.path.join(root, file), os.path.relpath(os.path.j
```

Part VII: Submission Score

In [3]:
```
# Specify the path to your JPEG image file
image_path = 'Score.jpg'

# Display the image
Image(filename=image_path)
```

Out[3]:



64	arunattri26	95.04482	2	2mo
65	pavan singu	118.19333	1	4d
66	Paul Marin	127.06468	2	7m
67	Ali Ahmadi 76	147.57939	1	20d
68	rafaelcalassara	256.41374	6	1mo
69	Razuki	296.36777	3	2mo
70	Scott M	322.74852	4	1d
71	Erik Lager	325.59229	1	1mo
72	Vieskov Yevhen	350.30333	1	14d
73	andywang947	361.40006	5	1mo
74	DELRUE Franck	364.78966	1	2mo
75	Team +1	684.06265	1	1mo